Datorteknik
AVR-assembler
Kodstil

Uppdateras löpande. Se datumet:

27 januari 2021

Innehåll

1 Inledning

Det här dokumentet beskriver formattering och upplägg hos AVR-assemblerkod.

Den givna kodstilen måste inte alltid följas till punkt och pricka men är en uppsättning tämligen hårda rekommendationer. Du har en viss ”konstnärlig frihet” med din kod men den friheten sträcker sig inte till att skriva onödigt komplicerad kod.

Kodreglerna är resultatet av åtskilliga års assemblererfarenhet med studenter.

1.1 Allmänt om koden

Tänk på att din kod skall kunna läsas, förstås och modifieras, kanske i flera år framöver.

> *Står valet mellan lättläst kod eller mindre kodmängd vinner alltid lättläst.*

1.2 Programrader

För att ge ett lättläst program skall varje programrad disponeras så att

- *Labels* börjar i kolumn 0 och på egen rad om inte läsbarheten försämras
- Instruktioner, `.equ`, `.org` och motsv börjar ett tab-stop in på raden
- Argument kan börja efter mellanslag eller ytterligare ett tab-stop
- Kommentarer utförs som

`;`	för kommentar till radslut
`//`	för kommentar till radslut
`/* */`	för längre block

  Kommentarer med ’`;`’ börjar senare i en bestämd kolumn, helst samma för alla kommentarer.

1.3 Struktur

Koden måste vara strukturerad.

- Det betyder här att den skall vara uppbyggd av *ändamålsenliga* subrutiner.
- Ägna särskild uppmärksamhet åt rutinernas namngivning, de måste ”göra vad de heter”.

- Det är fullt rimligt att koden består av flera subrutiner under programframtagning och felsökning som mot slutet slås ihop till färre rutiner för att snygga upp programflödet och ge tydligare kod. Läsbarheten är viktig.[1]
- Uppenbart repetitiva kodstycken skall loopas.
- Man kan dra det hela så långt att huvudprogrammet kanske enbart består av tre subrutin-anrop à la:

```
    call    INIT_HARDWARE
MAIN:
    call    GET_INPUT
    call    PROCESS
    jmp     MAIN
```

1.4 Ansvarsområde

Tänk på vilket *ansvarsområde* en rutin har, det skall framgå av rutinens namn (*label*).

En subrutin som heter `LINE_PRINT:` förväntas skriva ut en `LINE` och inget annat. Inför inte korsberoenden mellan olika rutiner. Programmera ortogonalt så att rutinerna är fristående enheter.

Låt varje rutin ha ett tydligt uppdrag och namnge den entydigt.

1.5 Subrutiner

Varje subrutin skall disponeras enligt följande ('|' ditsatta för att markera andra programrader)

```
EN_RUTIN:
    push    r16
    |
AAA:
    |
    jmp     BBB
CCC:
    call    DDD
    |
    rjmp    FFF
BBB:
    |
    breq    AAA
    rjmp    CCC
FFF:
    pop     r16
    ret
```

- Notera att hopp (`jmp`/`br*`) inte får ske *mellan* subrutiner, enbart *inom* en rutin.
- Notera också speciellt att varje subrutin har *en* väldefinierad ingång och endast ***en*** (1) utgång `ret`, och denna `ret` skall vara sista raden i subrutinen. Det är alltså inte tillåtet att göra returer från flera ställen i rutinen.[2]

[1] Kod som man kanske inte skulle kunnat ta fram om man skrev för stora oorganiserade kodstycken från början? Det blir en individuell uppfattning om tycke och smak. För många rutiner förvirrar, för få försvårar felsökning.

[2] Detta ger också mindre programstorlek i och med att uppsamlingskod (`pop` &c) enbart behöver förekomma på ett enda ställe per subrutin.

- Huvudflödet i en rutin skall vara nedåt.
- Håll ihop rutiner som har med varann att göra. Sprid inte kod överallt.
- Det är helt i sin ordning att ha korta subrutiner för ökad läslighet:

```
NEXT_BIT:
    lsl     r16
    ret
```

- Använd så få hopp som möjligt. Med bättre villkor kan det bli färre rader kod som här

```
    |                           |
    breq    DIT                 brne    BORT
    rjmp    BORT            DIT:
DIT:                            |
    |                           rjmp    EXIT
    rjmp    EXIT            BORT:
BORT:                           |
    |                       EXIT:
EXIT:                           ret
    ret
```

Med reglerna ovan undviker du att koden ”tarmar” ut sig, utan håller sig inom sin rutin.

1.6 Variabler

Använd variabler men minimera sidoeffekter genom att

- *Undvika* globala variabler.
- *Använda* lokala variabler där det krävs.
- *Använda* `.equ` och `def` där dessa är motiverade.
- Använd inte onödigt många register.
- *Använda* labels även i `.dseg`.
- *Använda* pekare där det är rimligt.
- *Använda* ASCII-tecken (`ldi r16,'A'`) i stället för hexkod för tydlighets skull.
- *Använda* `push` och `pop` men glöm inte variabler på stacken.
- Avbrottsrutiner och programmoduler får *i sin helhet* inte ha sidoeffekter.

1.7 Generalitet

Koden skall vara generell där så är möjligt.

Det är begränsande att skriva kod som bara kan göra AD-omvandling på en kanal om hårdvaran har stöd för flera. Skriv istället en rutin som tar kanalnumret som argument.

```
AD_READ:                ; in: channel i r16
    |                   ; out: measured value 0-255 i r16
    ret
```

Vill man sedan göra en rutin för omvandling av till exempel ett tempsensorvärde gör man detta i en egen rutin med lämpligt beskrivande namn:

```
READ_TEMP:              ; get temp in r16 (0-255 deg C)
    ldi     r16,3       ; sensor is channel 3
    call    AD_READ     ; read value
    call    VALUE_TO_C  ; convert to human readable form
    ret
```

Nu kan `AD_READ` användas för att läsa av joystickar och tryckknappar osv också.

1.8 Strukturera koden inom filen

Markera kodavsnitt och subrutiner som hänger ihop så de är lätta att hitta i kodmassan när man scrollar. Till exempel:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LCD-rutiner
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;
    ; LCD_OPEN
    ;
LCD_OPEN:
    |
    ret

    ;
    ; LCD_PRINTZ, print asciiz at Y
    ;
LCD_PRINTZ:
    |
    ret
```

1.9 Strukturera koden i flera filer

Lägg tidigare markerade kodavsnitt och subrutiner som hänger ihop i egna filer. Låt sedan inkludera dessa filer i huvudfilen. Detta förstärker kodens modularitet och medför ett tydligare fokus på sammanhängande rutiner som *programoduler*. Dessutom blir huvudfilen mindre och lättare att hantera.

Exempel: Kodavsnittet för en hypotetisk `HW`-drivrutin nedan kan klippas ut och läggas i separat fil.

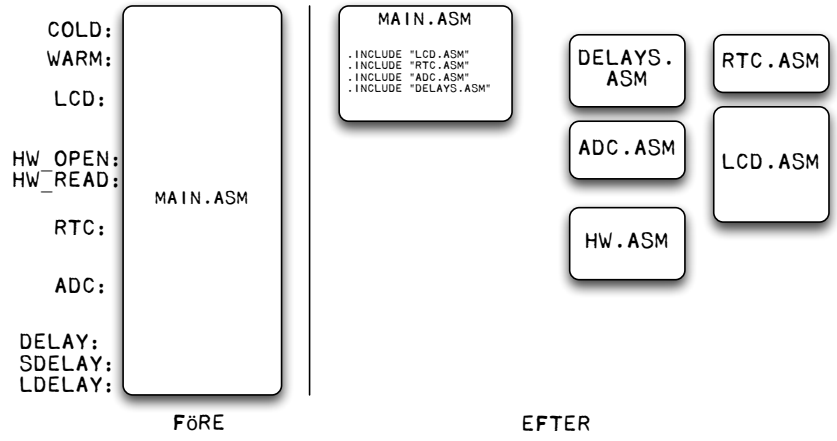
Drivrutinens publika rutiner är `HW_OPEN:` och `HW_READ:`. Internt beror dessa på `HW_HELP1:` och `HW_HELP2:`. De senare betraktas som drivrutinens *privata* rutiner.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; HW-rutiner
; HW_OPEN, HW_READ
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .dseg
HW_STATUS:      .byte   1
    .cseg
```

```
    ;
    ; HW_OPEN
    ;
HW_OPEN:
    |
    call    HW_HELP1
    |
    ret

    ;
    ; HW_READ
    ; In: address in r16
    ; Ut: value in r16
HW_READ:
    |
    call    HW_HELP2
    |
    ret

HW_HELP1:
    |
    ret

HW_HELP2:
    |
    ret
;--------------------------
```

Notera hur byten `HW_STATUS` allokeras i samma fil.

Man kan undvika onödig stackhantering genom att enbart låta de *publika* rutinerna skapa lokala register med `push`/`pop`. De privata rutinerna får då aldrig anropas utifrån.



Exempel: Man behöver en global variabel för en `BUSY`-flagga. Flaggan kan deklareras tillsamman med sina rutiner i separat assemblerfil:

```
;--------------------------
    .dseg
BUSY:       .byte   1
    .cseg

TOGGLE_BUSY:
    push    r16
    sts     r16,BUSY
    com     r16
    lds     BUSY,r16
    pop     r16
    ret
;--------------------------
```